CIS 121—Data Structures and Algorithms with Java—Fall 2015

Homework 8—Tries

Due: Thursday, November 19, 10:30am online

2 Required Problems (55 points), 1 Optional Extra Credit Problem (10 points), Questions (12 points), and Style and Tests (13 points)

DO NOT modify methods, method headers, or variables that were already provided to you.

DO NOT add public or protected methods.

DO introduce helper methods that are package-private (i.e., do not have a privacy modifier) as you deem necessary.

Motivation: Tree? Trie?

A prefix trie is an ordered tree data structure, which stores string keys by storing characters in nodes. The “prefix” part of the name comes from the fact that common prefixes between keys share the same path from the root in the trie. For a naive trie, is a fair amount of overhead, since every character lives in its own node. One example of an optimization that addresses this object overhead issue is a PATRICIA trie, in which each node that is an only child is merged with its parent. Unfortunately, you will not be implementing PATRICIA tries for this assignment.

TrieMap (40 points)

Files to submit: TrieMap.java, TrieMapTest.java

This should be a straightforward “standard” trie implementation, as described in the lecture notes. We have provided a skeleton for you in the form of a nested class Node and a few helper methods, and we’ve left you to worry about the following method stubs:

1. put(CharSequence key, V value)
2. get(CharSequence key)
3. containsKey(CharSequence key)
4. containsValue(V value)
5. remove(CharSequence key)
6. clear()

Each method stub contains further instructions. It is critical that you read both the Javadoc specification (best done by using Eclipse to read the spec) and the implementation comments for each method. They contain necessary information on behavior of these methods, hints on how to go about implementing them, and important explanations of differences from the HashMap implementation.

The trickiest part of this implementation will be correct removal of keys. You might find it helpful to work out examples of each method on paper before going forward with your implementation.

1The term “trie” comes from retrieval.
Extra Credit: Lazy iterator (10 points)

Implement the `entryIterator()` method to return the entries in lexicographic order with respect to the keys. You must write this as a true lazy iterator—an implementation that simply dumps all the elements into a collection and retrieves an iterator from the collection will be awarded no points. The iterator only stores enough state to do its job. Constraints:

- The running time must be linear in the number of elements in the trie.
- The space usage must be proportional to the height of the trie.

If this last constraint about space usage is too difficult, you can ignore that constraint for half credit, maximum 5 points.

Boggle (15 points)

The game of Boggle consists of an $n \times n$ square of letters from the English alphabet. The goal is to come up with all of the words that can be formed by chaining together orthogonally and diagonally adjacent letters from the grid. We will implement an easier version of this game where you need not worry about using only unique grid squares—that is, you will be solving the problem for where you can re-use grid squares when you search for a neighboring letter.

Take a moment now to think about a solution to this problem.

*Hint:* Since this problem is presented in a homework about tries, how might we leverage tries to solve this problem?

Please refer to the end of the writeup for a sketch of the solution.

Questions (12 points)

File to submit: `questions.pdf`

Please answer the following questions in a LaTeX’d document named `questions.pdf`:

1. Explain the space usage of your implementation of the Boggle algorithm. Explain the running time of your implementation of the Boggle algorithm.

2. What should be the optimal space usage and running time of the Boggle algorithm? If you did not meet the optimal characteristics, how would you fix your implementation to achieve them?

3. How might you adapt your algorithm if we do not permit squares in the grid to be reused? What would be the new space usage and running time? Does your answer for space usage change if we know that the size of any word will be greatly less than the total number of squares in the grid?

4. Examine the specification for `TrieMap.entryIterator()`. What kind of traversal needs to be performed by the iterator?

5. For the Boggle problem, do you think breadth-first search or depth-first search is better? Please point to specific qualities of BFS or DFS that suit a solution to the Boggle problem.
Style & Tests (13 points)

The above parts together are worth a total of 67 points. The remaining 13 points are awarded for code style, documentation and sensible tests. Style is worth 5 of the 13 points, and you will be graded according to the 121 style guide.

On this assignment in particular, focus on the behaviors that are specific to one implementation or the other. Especially consider edge cases, exceptions, and null keys and values. As always, use multiple methods instead of cramming a bunch of asserts into a single test method, and be sure to demonstrate that you’ve considered “bad” inputs, exceptions, etc. Your test cases will be manually graded by the TAs, and are worth 5 of the 10 points. You will have to thoroughly test your code to get full points! This includes testing any additional helper methods you have written. Note: you will not be able to write JUnit test cases for any private methods. Instead, make them package-private by leaving off any privacy modifier (i.e., do not write public or private).
Boggle!

Now that you’ve thought about the problem, here’s a skeletal solution:

Solution (breadth-first search). First, initialize the queue with all of valid one-letter prefixes in the grid as well as the node of the trie. Until the queue is empty, dequeue a prefix, check if extending the prefix with a letter from neighboring square produces a valid prefix, and enqueue this new prefix on the queue.

Solution (depth-first search). For each grid square containing a valid one-letter prefix, perform a depth-first search rooted from that square with the corresponding node of the trie. In the depth-first search routine, for each neighbor, check if extending the prefix with that neighboring letter produces a valid prefix, and recursively call the depth-first search routine on that neighboring square for the new prefix.

You are free to implement either solution. Be sure to work out an example out on paper before you start your implementation.

There are some details left out. Some important questions:

How do we efficiently determine if a prefix is valid? How do we know when we found a valid word? We will utilize a trie! We just need the node that the prefix ends on, along with the prefix itself. Then to check if we can extend the prefix with any of the neighboring squares, we need only check if the node has a child for that letter. This can be done efficiently if we have the particular node of the trie corresponding to the prefix in every iteration. Then checking if an extension is a valid prefix only involves checking if the corresponding node has a child for the next letter.

How do we store nodes representing a prefix? How do we handle what square the prefix ends on? This is just bookkeeping. For the breadth-first search, you might need some kind of internal object to go on the queue. For the recursive depth-first search, just pass any extra state that you might need.